Cloudproxy Nuts and Bolts

John Manferdelli, Tom Roeder, Kevin Walsh[1]

Overview

Cloudproxy is a software system that provides *remotely authenticated* isolation, confidentiality and integrity of code and data for Hosted Systems preventing attacks from co-tenants and (under modest assumptions) insiders in a remote data center on supporting hardware. To achieve this, Cloudproxy uses on two components: a "Host System" (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to a "Hosted System" (VM, Application, Container).

Cloudproxy provides a mechanism, at each level of the software stack, to isolate Hosted Systems, measure and remotely verify the exact software and configuration information constituting the Hosted System and provide security services like sealing that ensures that information (like keys) can be securely provisioned and retrieved only by the correct Hosted System, while isolated, on a supported platform.

A key concept for Cloudproxy is Code Identity and Measurement that is coupled with isolation and secret provisioning. A Host System measures a Hosted System incorporating the actual binary code and configuration information affecting execution resulting in an unforgeable, compact global identity for that code and execution context. Since the Hosted System knows the "identity" of each Hosted System (i.e.- the unforgeable global identity), it can store secrets that only the Hosted System will receive[2]. The Host Systems can also "attest" to statements made by Hosted Systems by incorporating the unforgeable global identity in statements it signs (again with keys only an isolated Host System has access to). The upshot of this is that a Cloudproxy Hosted System can be isolated, maintain secrets only it knows to encrypt and integrity protect all data it receives or sends, and it can securely authenticate itself over an otherwise unprotected network connection and thus employ authenticated public keys tied to its identity that can be relied upon by communicating parties.

Readers can consult [1] for a fuller description. Source code is in [2].

The Tao Library and API

A Hosted System uses the Cloudproxy API, called the Tao, to achieve the security promises (program isolation, and confidentiality and integrity for programs and data) provided by Cloudproxy. The programming model is simple and requires only a few API calls. The Tao provides a programming interface in Go or C++.

---

[1] John is manferdelli@google.com, Tom is tmroeder@google.com and Kevin Walsh is kevin.walsh@holycross.edu.

[2] To do this, the Host System must be isolated and have access to secrets only it knows. The foundation for this consists of primitives hardware provides to the "base" Cloudproxy systems it boots.

The basic Tao API calls are:

AddHostedProgram instructs the Host System to measure and start a new, isolated Hosted System. It names the binary image and other context data used to start the program. The Hosted System could be, for example, a VM if the Host System is a VMM or an isolated Linux process if the Host System is Linux.

Seal takes an opaque data blob and appends the measurement of the Hosted System. It encrypts and integrity protects the resulting object (using keys only the Host System knows) and returns the resulting opaque object to the Hosted System. Hosted Systems typically "seal" private signing and encryption keys so they can be later recovered when the Hosted System is restarted using "Unseal" below.

Unseal takes an opaque blob (produced by a prior "Seal") from a Hosted System. It decrypts (and checks the integrity of) the blob and compares the measurement of the Hosted System requesting the unseal with the measurement of the Hosted System named in the blob. If the measurements match, it returns the protected data.

Attest takes a blob from a Hosted System and signs a statement naming the blob and the measurement of the Hosted System requesting the Attest. It returns the signed statement (the "Attestation") along with a certificate (the "Host Certificate") from an authority certifying that the public key it used to sign the statement belongs to a verified Host System with enumerated security characteristics[3]. The meaning of the signed blob is, informally, "*Statement X came from the program with Measurement M while it was isolated*." Hosted Systems mainly use attest as follows: The Hosted System generates a public-private key pair and seal the private key. Then they request an Attestation of the corresponding public key. A party receiving the Attestation and Host Certificate can cryptographically verify the public key came from the named program while isolated and thus subsequent proof of possession of the private key can be used to authenticate statements from the Hosted System.

GetRandom provides cryptographically random bits, typically for key generation.

Principal Names

Principal names in Cloudproxy are hierarchical and securely name the principal. For example, a principal rooted in a public key will have the public key (or a cryptographic hash of it) in its name and a program principal (a measured Cloudproxy Hosted System) will have the measurement in the principal name (i.e.-a cryptographic hash).

The root name for a hosted program, in the development case, might look something like

[3] See [1] for details for the "Trust Model" enabling a recipient of such a certificate to rely on the association between the public key named in the Host Certificate and a trustworthy Cloudproxy Hosted System.

```
key([0801100118012245080112410 46cdc82f70552eb...]).Program([25fac93bd4c
c868352c78f4d34df6d2747a17f85...])
```

Here, `key([0801100118012245080112...])` represents the signing key of the host and `Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...])` extends the host name with the hash of the Hosted System. If the host were a Linux host rooted in a TPM boot, its name would include the AIK and the PCRs of the booted Linux systems, the hash of the Authenticated Code Module ("ACM") that initiated the authenticated boot and the hash of the Linux image and it's initramfs[4]. In the section on SimpleExample execution output, there are many more examples of Tao Principal names.

The Tao Paradigm

The Tao is almost always used in a stereotypical way which we refer to as the Tao Paradigm. Cloudproxy programs always have policy public keys embedded (PK_{policy}) in their image either explicitly or implicitly. Statements signed by the corresponding private key (pK_{policy}), and only those statements, are accepted as authoritative and acted on by these programs. The policy key(s) plus the Hosted System code and configuration, reflected in its measurement, fully describe how the Hosted System should behave and, hence, an authenticated measurement is a reliable description of expected behavior.

In the Tao Paradigm, when a program first starts on a Hosted System, it makes up a public/private key-pair ($PK_{program}$/ $pK_{program}$) and several symmetric keys that it uses to "seal" information for itself. A Hosted System then "seals," using the Host System interface, all this private (key) information[5]. After that, the Hosted System requests an Attestation from its Host System, naming the newly generated $PK_{program}$ and sends the resulting Attestation to a security domain service which confirms the security properties in the Attestation and Host Certificate[6]. If the Attestation and Host Certificate meet security domain requirements, the security domain service signs (with pK_{policy}) an x509 certificate specifying $PK_{program}$ and the Tao Principal Name of the Hosted System. The resulting certificate, called the *Program Certificate*, can be used by any Hosted System to prove its identity to another Hosted System in the same security domain. Program Certificates are used to negotiate encrypted, integrity protected SSL-like channels between Hosted Systems (the "Tao Channel"); Hosted System can share information over these channels with full assurance of the code identity and security properties of its channel peer. Once established, each endpoint of the Tao Channel "speaks for" its respective Hosted System.

Hosted Systems in the same security domain can fully trust other authenticated Hosted Systems in that security domain with data or processing. Typically, a Hosted System uses the symmetric keys it generates and seals at initialization to encrypt and integrity protect information it stores on disks or remotely.

---

[4] Initramfs will have security critical code like the service that implements the Tao so it must be measured along with the kernel image to provide an accurate identity for the "running Linux OS."

[5] The Tao also provides rollback protection for this sealed data.

[6] The actual attestation being signed by the Host System expressed in a formalized language is $PK_{program}$ speaksfor the Hosted System Principal name.

Employing a centralized security domain service eliminates the need for each and every Cloudproxy Hosted System in a security domain to maintain lists of trusted hardware or trusted programs and simplifies distribution, maintenance and upgrade.

Often, Hosted Systems in the same security domain will share intermediate keys used to protect data that may be used on many Host System environments. As discussed below, when software is upgraded or a new Hosted System in a security domain is added, these keys can be shared based on policy-key signed directives as Host or Hosted Systems are upgraded or new systems are introduced in a controlled but flexible way eliminating the danger that data might become inaccessible if a particular Cloudproxy system is replaced or becomes damaged or unavailable.

The Extended Tao

Given the Tao Paradigm that is almost universally employed, the Tao contains some additional support functions. All these functions are in the Tao Library (along with the basic Tao Library functions above).

DomainLoad is used to store and retrieve Program Certificates and sealed policy data.

GetTaoName gets the principal name for the hosted system.

GetSharedSecret Tom?

Parent gets the parent interface to the Tao.

ExtendTaoName allows a Hosted System to extend its Principal Name with arbitrary data. For example, rather than having a policy embedded in a program image, a Hosted System can extend its name with a policy key it reads and the new Principal Name will reflect this value.

The API for the Tao Library is set forth in the appendix but it is most easily learned by looking at the code example and corresponding output below.

The Tao Library also contains helper functions to build and verify Program Certificates, perform common crypto tasks like key generation and establish the Tao Channel. Again, these are most easily understood by looking at the code example below.

Finally, the Tao Library has rather extensive and flexible authorization support. Authorization decision are performed by *guards* make.

Current guards include:

- The liberal guard: this guard returns true for every authorization query
- The conservative guard: this guard returns false for every authorization query

- The ACL guard: this guard provides a list of statements that must return true when the guard is queried for these statements.
- The datalog guard (used in the example below): this guard translates statements in the CloudProxy auth language (see tao/auth/doc.go for details) to datalog statements and uses the Go datalog engine from github.com/kevinawalsh/datalog to answer authorization queries. See install.sh for an example policy.

Hardware roots of Trust

Cloudproxy requires that the lowest level system software (the “base system) be measured by a hardware component which also provides attest services and seal/unseal services and some hardware assist to isolate of Hosted Systems. Absent hardware protection, remote users have no principled way to trust the security promises (isolation, confidentiality, integrity, verified code identity) since “insiders” might silently change security critical software or steal keys.

Cloudproxy supports TPM 1.2 and TPM 2.0 as hardware roots of Trust for Host Systems booted on raw hardware. We have implemented support for other mechanisms and believe adding a new hardware mechanism is relatively. In addition, Cloudproxy also can initialize and run on a “soft Tao” which simulates secure base system protection (but is not secure). This allows for easy debugging.

Once the base Host System is safely measured and booted on a supported hardware, Cloudproxy implements support for recursive Host Systems at almost every layer of software including:

1. A Host System consisting of hardware (e.g. - TPM, SMX) that hosts a VMM which isolates Hosted Systems consisting of Virtual Machines.
2. A Host System running in an operating system which isolates Hosted Systems consisting of processes (or applications).
3. A Host System running in an operating system which isolates Hosted systems hosted consisting of subordinate Operating Systems or Containers.
4. A Host System running in an application (like a browser) which isolates Hosted Systems consisting of sub-applications, like plug-ins.

In all cases, Hosted Systems have the same Tao interface to the Host System and can use any non-Cloudproxy host service (for example, any system call on Linux) so the programming model at each Hosted System layer is essentially unchanged from the non-Cloudproxy case.

Sample Applications

This paper is intended to allow you to use Cloudproxy immediately on a Linux based Cloudproxy Host System. To this end we include installation instructions for running under a “soft Tao” and TPM 1.2 based Tao protected hardware with SMX extensions. We also include the annotated code for a simple application called, cleverly, SimpleExample.

There are more complex examples in go/apps.

Installing Cloudproxy

First, you should download the Cloudproxy repository from [2]. To do this, assuming you have git repository support, type

Git clone https://github.com/jlmucb/cloudproxy.

You can also download a zipped repository at the same address. You should probably install this in ~/src/github.com/jlmucb (which we refer to as $CLOUDPROXYDIR) to save go compilation problems later. It's also a good idea to put go binaries in ~/bin as is common. Follow the installation instructions in $CLOUDPROXYDIR/Doc. That directory also contains [1] and an up to date version of this document as well as installation instructions for TPM 2.0 capable machines and installation for a Cloudproxy enabled KVM hypervisor and DOcker containers.

You must also install the Go development tools (and C++ development tools if you use the C++ version) as well as protobuf, gtest and gflags as described in the Go documentation.

Next, compile, and initialize the SimpleExample application in $CLOUDPROXYDIR/go/apps/SimpleExample and run it as described in the next section.

Simple Example

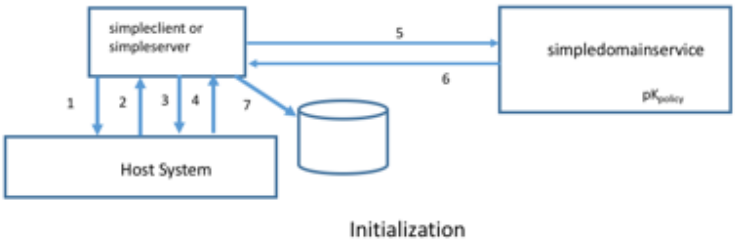
There are three application components in SimpleExample, each producing a separate executable:

1. Simple Client (in $CLOUDPROXYDIR/go/apps/SimpleExample/SimpleClient simpleclient.go)
2. Simple Server (in $CLOUDPROXYDIR/go/apps/SimpleExample/SimpleServer/simpleserver.go)
3. Simple Security Domain Signing Service (in $CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomainService/simpledomainservice.go)

Common code used by the client and server is in $CLOUDPROXYDIR/go/apps/SimpleExample/taosupport.

The *simpleserver* makes up a secret waits for *simpleclient* to request their secret. Each *simpleclient* uses a Tao Channel to contact the *simpleserver* to learn the secret. We don't implement rollback protection or distributed key management for intermediate secrets in SimpleExample just to keep the example as simple as possible. *SimpleDomainService* is the domain service for SimpleExample. When *simpleclient* and *simpleserver* start for the first time on the Host System, they provide Attestations to *simpledomainservice* (as described above)

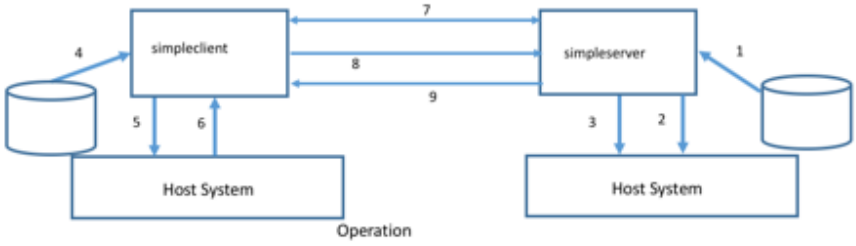
and, if the measurements are correct, *simpledomainservice* signs their respective Program Certificates with pK_{policy}). This interaction is depicted below.



Initialization

Initialization

1. Simpleclient (or simpleserver) generates public/private key pair $PK_{simpleclient}$, $pK_{simpleclient}$. Simpleclient requests Host System attest $PK_{simpleclient}$.
2. Host System returns attestation
3. Simpleclient generates additional symmetric keys and request Host System seal symmetril keys and $pK_{simpleclient}$.
4. Host System returns sealed blobs.
5. simpleclient connects to simpledomainservice and transmits attestation.
6. Simpledomainservice returns signed Program Certificate.
7. Simpleclient stores sealed blobs and Program Certificate for later activations.



Operation

Operation

1. Simpleserver reads previous sealed blobs and Program Certificate.
2. Simpleserver requests Host System unseal blobs yielding symmetric keys and private program key.
3. Host system returns unsealed blobs.
4. Simpleclient reads previous sealed blobs and Program Certificate.
5. Simpleclient requests Host System unseal blobs yielding symmetric keys and private program key.
6. Host system returns unsealed blobs.
7. Simpleclient and simpleserver open encrypted, integrity protected channel using their program keys and certificates.
8. Simpleclient transmits a request to retrieve secret.
9. Simpleserver retruns secret.

We describe the Tao API, compilation and installation, execution and output of the Go version of SimpleExample in the sections below. We also provide annotation for all the SimpleExample code containing all the critical Cloudproxy elements to help you get used to the programming model. Since the domain service does not use Tao primitives directly, we don't annotate that code here although `$CLOUDPROXYDIR/go/apps/SimpleExample` contains a full working version. A corresponding version of the annotated C++ version appears in Appendix III.

Although SimpleExample is very simple, the Tao relevant code in SimpleExample can be used with little change even in complex Cloudproxy applications.

The API – Go

Domains represent security contexts. Security contexts encapsulate configuration information like names, path to key blobs, path to policy key, and the guard employed for authorization decisions.

CreateDomain initializes a new Domain, writing its configuration files to a directory. This creates the directory and, if needed, a policy key pair encrypted with the given password when stored on disk; it also initializes a default guard. The call is:

```
func CreateDomain(cfg DomainConfig, configPath string, password []byte) (*Domain, error)
```

Any parameters left empty in cfg will be set to reasonable default values.

Domain information is loaded from a text file, typically called tao.config via the call:

```
LoadDomain(configPath string, password []byte)(*Domain, error)
```

which returns a domain object, if successful. The password is used to load a key set from disk. If no password is provided, then *LoadDomain* will attempt to load verification keys only. For example, *LoadDomain* is called with a configPath and an empty password to load the policy verification key.

The network interface for the Tao channel consists of:

- `func DialWithKeys(network, addr string, guard tao.Guard, v *tao.Verifier, keys *tao.Keys) (net.Conn, error)`
- `func Listen(network, laddr string, config *tls.Config, g tao.Guard, v *tao.Verifier, del *tao.Attestation) (net.Listener, error)`

The complete Tao Library API is described in Appendix II but the best way to learn it is by looking at the annotated code here and in the distribution.

Simple Client in Go (annotated)

The *simpleclient* code (omitting imports and flag definitions) is as follows:

```
func main() {

        // This holds the cloudproxy specific data for simpleclient
        // including the Program Cert and Program Private key.
        var clientProgramData taosupport.TaoProgramData

        // Make sure we zero keys when we're done.
        defer taosupport.ClearTaoProgramData(&clientProgramData)

        // Parse flags
        flag.Parse()
        serverAddr = *serverHost + ":" + *serverPort

        // Load domain info for this domain and establish Clouproxy keys and properties.
        // This handles reading in existing (sealed) Cloudproxy keys and properties, or,
        // if this is the first call (or a call after state has been erased), this also
        // handles initialization of keys and certificates with a domain server holding
        // the private policy key.
        // If TaoParadigm completes without error, clientProgramData contains all the
        // Cloudproxy information needed throughout program execution and, in addition,
        // ensures that this information is sealed and stored in simpleClientPath for
        // subsequent invocations.
        if taosupport.TaoParadigm(simpleCfg, simpleClientPath, &clientProgramData) !=
                        nil {
                log.Fatalln("simpleclient: Can't establish Tao")
        }
        fmt.Printf("simpleclient: TaoParadigm complete, name: %s\n",
           clientProgramData.TaoName)

        // Open the Tao Channel using the Program key.  This program does all the
        // standard channel negotiation and presents the secure server name after
        // negotiation is complete.  ms is the bi-directional confidentiality and
        // integrity protected channel between simpleclient and simpleserver.
        ms, serverName, err := taosupport.OpenTaoChannel(&clientProgramData, &serverAddr)
        if err != nil {
                log.Fatalln("simpleclient: Can't establish Tao Channel")
        }
        log.Printf("simpleclient: establish Tao Channel with %s, %s\n",
                serverAddr, serverName)
```

```
        // Send a simple request and get response.
        // simpleclient and simpleserver.  There's only on request: tell me the
        // secret.
        secretRequest := "SecretRequest"

        msg := new(taosupport.SimpleMessage)
        msg.RequestType = &secretRequest
        taosupport.SendRequest(ms, msg)
        if err != nil {
                log.Fatalln("simpleclient: Error in response to SendRequest\n")
        }
        respmsg, err := taosupport.GetResponse(ms)
        if err != nil {
                log.Fatalln("simpleclient: Error in response to GetResponse\n")
        }

        // This is the secret.
        retrieveSecret := respmsg.Data[0]

        // Encrypt and store the secret in simpleclient's save area.
        out, err := taosupport.Protect(clientProgramData.ProgramSymKeys, retrieveSecret)
        if err != nil {
                log.Fatalln("simpleclient: Error protecting data\n")
        }
        err = ioutil.WriteFile(path.Join(*simpleClientPath,
                        "retrieved_secret"), out, os.ModePerm)
        if err != nil {
                log.Fatalln("simpleclient: error saving retrieved secret\n")
        }

        // Close down.
        log.Printf("simpleclient: secret is %s, done\n", retrieveSecret)
}
```

Simple Server in Go

Here is the *simpleserver* code without imports, flag definitions, support code, and code that is similar to *simpleclient.* Error processing has also been removed for brevity.

```
// Handle service request, req and return response over channel (ms).
// This handles the one valid service request: "SecretRequest"
// and terminates the channel after the first successful request
// which is not generally what would happen in most channels.
// Note that in the future, we might want to use grpc rather than custom
// service request/response buffers but we don't want to introduce complexity
// into this example.  The single request response buffer is defined in
// taosupport/taosupport.proto.
func HandleServiceRequest(ms *util.MessageStream,
      serverProgramData *taosupport.TaoProgramData,
      clientProgramName string, req *taosupport.SimpleMessage) (bool, error) {

        // The somewhat boring secret is the corresponding simpleclient's program name||43
        secret := clientProgramName + "43"

        if *req.RequestType == "SecretRequest"  {
                req.Data = append(req.Data, []byte(secret))
                …
                return true, nil
        } else {
                …
        }
}

// This just handles the requests.
func serviceThead(ms *util.MessageStream, clientProgramName string,
```

```
        serverProgramData *taosupport.TaoProgramData) {

        for {
                req, err :=  taosupport.GetRequest(ms)
                …
                …
                terminate, _ := HandleServiceRequest(ms, serverProgramData,
                        clientProgramName, req)
                if terminate {
                        break
                }
        }
        …
}


// This is the server. It implements the server Tao Channel negotiation corresponding
// to the client's taosupport.OpenTaoChannel.  It's possible we should move this into
// taosupport/taosupport.go since it should not vary very much from implementation to
// implementation.
func server(serverAddr string, serverProgramData *taosupport.TaoProgramData) {

        var sock net.Listener

        // Set up the single root certificate for channel negotiation which is the
        // policy key cert.
        pool := x509.NewCertPool()
        policyCert, err := x509.ParseCertificate(serverProgramData.PolicyCert)
        if err != nil {
                log.Printf("simpleserver, can't parse policyCert: ", err, "\n")
                return
        }
        // Make the policy cert the unique root of the verification chain.
        pool.AddCert(policyCert)
        tlsc, err := tao.EncodeTLSCert(&serverProgramData.ProgramKey)
        …
        conf := &tls.Config{
                …
        }

        // Listen for clients.
        …
        sock, err = tls.Listen("tcp", serverAddr, conf)
        …
        // Service client connections.
        for {
                log.Printf("server: at accept\n")
                conn, err := sock.Accept()
                …
                var clientName string
                err = conn.(*tls.Conn).Handshake()
                …
                peerCerts := conn.(*tls.Conn).ConnectionState().PeerCertificates
                …
                peerCert := conn.(*tls.Conn).ConnectionState().PeerCertificates[0]
                …
                clientName = peerCert.Subject.OrganizationalUnit[0]
                log.Printf("server, peer client name: %s\n", clientName)
                ms := util.NewMessageStream(conn)

                // At this point the handshake is complete and we fork a service thread
                // to communicate with this simpleclient.  ms is the bi-directional
                // confidentiality and integrity protected channel corresponding to the
                // channel opened by OpenTaoChannel.
                go serviceThead(ms, clientName, serverProgramData)
        }
}

func main() {

        // main is very similar to the initial parts on main in simpleclient.
```

```
        // see the comments there.
        …

        server(serverAddr, &serverProgramData)
        log.Printf("simpleserver: done\n")
}
```

Some Common code in Go

Here is the important common code. We only include critical code.

```
type TaoProgramData struct {
        // true after initialization.
        Initialized       bool
        // Program name.
        TaoName           string
        // DER encoded policy cert for domain.
        PolicyCert        []byte
        // Private program key.
        ProgramKey        tao.Keys
        // Symmetric Keys for program.
        ProgramSymKeys    []byte
        // Program Cert.
        ProgramCert       []byte
        // Path for program to read and write files.
        ProgramFilePath   *string
}

func ClearTaoProgramData(programData *TaoProgramData) {
        …
}

// RequestTruncatedAttestation connects to a CA instance, sends the attestation
// for an X.509 certificate, and gets back a truncated attestation with a new
// principal name based on the policy key.
func RequestDomainServiceCert(network, addr string, keys *tao.Keys,
                v *tao.Verifier) (*tao.Attestation, error) {
        …
        tlsCert, err := tao.EncodeTLSCert(keys)
        …
        conn, err := tls.Dial(network, addr, &tls.Config{
                RootCAs:            x509.NewCertPool(),
                Certificates:       []tls.Certificate{*tlsCert},
                InsecureSkipVerify: true,
        })
        …

        // Tao handshake: send client delegation.
        ms := util.NewMessageStream(conn)
        …

        // Read the truncated attestation and check it.
        var a tao.Attestation
        …
        ok, err := v.Verify(a.SerializedStatement, tao.AttestationSigningContext,
a.Signature)
        …
        if !ok {
                return nil, errors.New("invalid attestation signature from Tao CA")
        }
        return &a, nil
}

func InitializeSealedSymmetricKeys(filePath string, t tao.Tao, keysize int) (
                []byte, error) {

        // Make up symmetric key and save sealed version.
        log.Printf("InitializeSealedSymmetricKeys\n")
```

```
        unsealed, err := tao.Parent().GetRandomBytes(keysize)
        …
        sealed, err := tao.Parent().Seal(unsealed, tao.SealPolicyDefault)
        …
        ioutil.WriteFile(path.Join(filePath, "sealedsymmetricKey"), sealed, os.ModePerm)
        return unsealed, nil
}

func InitializeSealedProgramKey(filePath string, t tao.Tao, domain tao.Domain) (
                *tao.Keys, error) {

        k, derCert, err := CreateSigningKey(t)
        …

        // Request attestations.  Policy key is verifier.
        na, err := RequestDomainServiceCert("tcp", *caAddr, k, domain.Keys.VerifyingKey)
        …
        k.Delegation = na
        pa, _ := auth.UnmarshalForm(na.SerializedStatement)
        var saysStatement *auth.Says
        if ptr, ok := pa.(*auth.Says); ok {
                saysStatement = ptr
        } else if val, ok := pa.(auth.Says); ok {
                saysStatement = &val
        }
        sf, ok := saysStatement.Message.(auth.Speaksfor)
        if ok != true {
                return nil, errors.New("InitializeSealedProgramKey: says doesnt have
speaksfor message")
        }
        kprin, ok := sf.Delegate.(auth.Term)
        if ok != true {
                return nil, errors.New("InitializeSealedProgramKey: speaksfor message
doesn't have Delegate")
        }
        newCert := auth.Bytes(kprin.(auth.Bytes))
        k.Cert, err = x509.ParseCertificate(newCert)
        …
        programKeyBlob, err := tao.MarshalSignerDER(k.SigningKey)
        …
        sealedProgramKey, err := t.Seal(programKeyBlob, tao.SealPolicyDefault)
        …
        err = ioutil.WriteFile(path.Join(filePath, "sealedsigningKey"), sealedProgramKey,
os.ModePerm)
        …
        err = ioutil.WriteFile(path.Join(filePath, "signerCert"), newCert, os.ModePerm)
        …
        delegateBlob, err := proto.Marshal(k.Delegation)
        …
        err = ioutil.WriteFile(path.Join(filePath, "delegationBlob"), delegateBlob,
os.ModePerm)
        …
        return k, nil
}

func (pp *TaoProgramData) FillTaoProgramData(policyCert []byte, taoName string,
                programKey tao.Keys, symKeys []byte, programCert []byte,
                filePath *string) bool {
        pp.PolicyCert = policyCert
        pp.TaoName = taoName
        pp.ProgramKey = programKey
        pp.ProgramSymKeys = symKeys
        pp.ProgramCert = programCert
        pp.ProgramFilePath = filePath
        pp.Initialized = true
        return true
}

// Load domain info for the domain and establish Clouproxy keys and properties.
// This handles reading in existing (sealed) Cloudproxy keys and properties, or,
// if this is the first call (or a call after state has been erased), this also
```

```
// handles initialization of keys and certificates including interaction with the
// domain signing service and storage of new sealed keys and certificates.
// If TaoParadigm completes without error, programObject contains all the
// Cloudproxy information needed throughout the calling program execution
// ensures that this information is sealed and stored for subsequent invocations.
func TaoParadigm(cfg *string, filePath *string,
               programObject *TaoProgramData) (error) {

        // Load domain info for this domain.
        simpleDomain, err := tao.LoadDomain(*cfg, nil)
        …

        // Get policy cert.
        if simpleDomain.Keys.Cert == nil {
                return errors.New("TaoParadigm: Can't retrieve policy cert")
        }
        derPolicyCert := simpleDomain.Keys.Cert.Raw
        …

        // hash of policyCert identifies the extension
        policyKeyName := sha256.Sum256(derPolicyCert)
        hexPolicyCert :=  hex.EncodeToString(policyKeyName[0:32])
      // Extend my Tao Principal name with policy key.
        t := make([]auth.Term, 1, 1)
        t[0] = auth.TermVar(hexPolicyCert)
        e := auth.PrinExt{Name: "key",
                          Arg: t}
        err = tao.Parent().ExtendTaoName(auth.SubPrin{e})
        …

        // Retrieve extended name.
        taoName, err := tao.Parent().GetTaoName()
        …
        log.Printf("TaoParadigm: my name is %s\n", taoName)

        // Get my keys and certificates.
        sealedSymmetricKey, sealedProgramKey, programCert, delegation, err :=
                LoadProgramKeys(*filePath)
        …
        // Unseal my symmetric keys, or initialize them.
        var symKeys []byte
        var policy string
        if sealedSymmetricKey != nil {
                symKeys, policy, err = tao.Parent().Unseal(sealedSymmetricKey)
                …
        } else {
                symKeys, err = InitializeSealedSymmetricKeys(*filePath, tao.Parent(),
                        SizeofSymmetricKeys)
                …
                }
        }
…
        // Get my Program private key if present or initialize it.
        var programKey *tao.Keys
        if sealedProgramKey != nil {
                programKey, err = SigningKeyFromBlob(tao.Parent(),
                        sealedProgramKey, programCert, delegation)
                …
        } else {
                // Get Program key.
                programKey, err = InitializeSealedProgramKey(
                        *filePath, tao.Parent(),
                        *simpleDomain)
                …
        }
        log.Printf("TaoParadigm: Retrieved Signing key\n")

        // Initialize Program policy object.
        ok := programObject.FilTaoProgramData(derPolicyCert, taoName.String(),
                *programKey, symKeys, programKey.Cert.Raw, filePath)
```

```
        if !ok {
                return errors.New("TaoParadigm: Can't initialize TaoProgramData")
        }
        return nil
}

// Establishes the Tao Channel for a client and returns the stream for subsequent reads
// and writes as well as the server's Tao Principal Name.
func OpenTaoChannel(programObject *TaoProgramData, serverAddr *string) (
                *util.MessageStream, *string, error) {

        // Parse policy cert and make it the root of our heierarchy for verifying
        // Tao Channel peer.
        policyCert, err := x509.ParseCertificate(programObject.PolicyCert)
            …
        pool := x509.NewCertPool()
        pool.AddCert(policyCert)

        // Open the Tao Channel using the Program key.
        tlsc, err := tao.EncodeTLSCert(&programObject.ProgramKey)
        …
        conn, err := tls.Dial("tcp", *serverAddr, &tls.Config{
                RootCAs:            pool,
                Certificates:       []tls.Certificate{*tlsc},
                InsecureSkipVerify: false,
        })
        …

        // server name.
        peerName := policyCert.Subject.OrganizationalUnit[0]

        // Stream for Tao Channel.
        ms := util.NewMessageStream(conn)
        return ms, &peerName, nil
}

…


// Returns sealed symmetric key, sealed signing key,
//   DER encoded program cert, delegation, error.
// Only returns errors if file exists but can't be read.
func LoadProgramKeys(filePath string) ([]byte, []byte, []byte, []byte, error) {
        …
        return sealedSymmetricKey, sealedProgramKey, derCert, ds, nil
}
```

Configuring, compiling and running SimpleExample

When the Tao Host System starts, it requires three kinds of information:

- A public key that roots the *Tao* on the hardware,
- Host data,
- Domain data (in our case for the simpleexample domain) including the policy key and corresponding private key, and the self signed policy cert.

In addition, we need an implementation for the "Host System." In our case, the Host System is Linux and the implementation (whether using a soft tao or a TPM) is linux_host.

The public key rooting the hardware tao is usually produced by a tpm utility; in the TPM 1.2 nomenclature, this is called the AIK. The public key rooting the TPM 2.0 is the endorsement key. In our demo, we use a "soft tao" which is rooted in a key.

The Host Data consists of keys and Host Certificate (used to validate nested Host System Attestation) and are in

`$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample/linux_tao_host`.

The Domain data including the policy key and corresponding private key, hostname, and information related to the guards used[7] as well as signatures over the binaries that are part of the domain. In our case, these are the *simpleclient* and *simpleserver* binaries.

In simpleexample, all these information files in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample`. Other sub-directories of `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample`, namely, `SimpleClient`, `SimpleServer` and `SimpleDomainService` contains data files stored and retrieved by these programs (like sealed keys and Program Certificates).

There is a single utility, called *tao* which initializes this domain data, activates the tao host and runs the applications. We provide shell scripts to call *tao* with the right arguments, these scripts are in SimpleDomain.

The scripts use several path variables, namely:

```
TAO_HOST_DOMAIN_DIR=~/src/github.com/jlmucb/cloudproxy/go/apps/simpleexample/SimpleDomain
OLD_TEMPLATE=$TAO_HOST_DOMAIN_DIR/domain_template.simpleexample
DOMAIN=/Domains/domain.simpleexample
TEMPLATE=/Domains/domain_template.simpleexample
BINPATH=~/bin
```

In addition, we need a generic domain template. We have provided a sample template in *SimpleDomain/domain_template.simpleexample*. However, you can generate such a template by running *gentemplate*, which consists of:

```
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -pass "xxx"
/home/jlm/src/github.com/jlmucb/cloudproxy/go/run/scripts/domain_template.pb > $TEMPLATE
sed "s/REPLACE_WITH_DOMAIN_GUARD_TYPE/Datalog/g"
```

This template contains information included in the policy cert, the basic datalog rules used by the domain when authenticating images and the location of the images which must me measured and included in the policy database in SimpleDomainService.

First, we must initialize the directory that will hold domain information. We do this by first

```
mkdir /Domains
```

and then calling initidomainstorage which consists of:

```
#
source ./defines
if [ -e $DOMAIN ]
then
  ls -l $DOMAIN
else
  mkdir $DOMAIN
fi
cp $OLD_TEMPLATE $TEMPLATE
source ./defines
if [[ -e $DOMAIN/SimpleClient]]
then
  echo "$DOMAIN/SimpleClient exists"
else
```

[7] Look at domain.go for further details.

```
  mkdir $DOMAIN/SimpleClient
  echo "$DOMAIN/SimpleClient created"
fi
if [[ -e $DOMAIN/SimpleServer]]
then
  echo "$DOMAIN/SimpleServer exists"
else
  mkdir $DOMAIN/SimpleServer
  echo "$DOMAIN/SimpleServer created"
fi
if [[ -e $DOMAIN/SimpleDomainService]]
then
  echo "$DOMAIN/SimpleDomainService exists"
else
  mkdir $DOMAIN/SimpleDomainService
  echo "$DOMAIN/SimpleDomainService created"
fi
```

To initialize the (soft) key, call initkey which does the following:

```
#
source ./defines
if [[ -e $DOMAIN/linux_tao_host ]]
then
  echo "$DOMAIN/linux_tao_host exists"
else
  mkdir $DOMAIN/linux_tao_host
  echo "$DOMAIN/linux_tao_host created"
fi
KEY_NAME="$($BINPATH/tao domain newsoft -soft_pass xxx -config_template $TEMPLATE
$DOMAIN/linux_tao_host)"
      echo "host_name: \"$KEY_NAME\"" >> $TEMPLATE
```

"`newsoft`" means generate a new soft key. The arguments following the flags "`-config_template -tao_ -pass`" specify respectively the location of the template, the location where the domain information is stored and the password protecting the private policy key. This produces the xxx file containing root Tao key. If using the tpm, you'd call a corresponding program to put the AIK in template.

To initialize the domain, call initdomain which does the following:

```
#
source ./defines
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -pub_domain_address
"127.0.0.1" -pass xxx
$BINPATH/tao domain policy -add_host -add_programs -add_linux_host -add_guard -tao_domain \
      $DOMAIN -pass xxx -config_template $TEMPLATE
```

The first call produces the files in $DOMAIN/linux_tao_host/{cert,keys,host.config}. The second measures the applications in the domain.

To initialize the (Linux) host, call inithost which does the following:

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -root -pass xxx
```

This generates linux host configuration information which is in SimpleDomain/domain.simpleexample/linux_tao_host. The argument to the "`-hosting`" flag is the kind of child hosts, namely, Linux processes. The "`-root`" flag means this is a "root" host (i.e. – the lowest level tao). For hosts stacked on other hosts, we would use the "`-stacked`" flag. For example,

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -stacked -parent_type tpm
```

To run the host, call runhost, which consists of:

```
$BINPATH/tao host start -tao_domain $DOMAIN -host linux_tao_host/ -pass xxx &
```

The argument to the "`-host`" flag is the subdirectory of SimpleDomain/domain.simpleexample that contains the host information.

Finally, to run a Hosted System, like simpleclient, we would say:

```
$BINPATH/tao run $BINPATH/simpleclient -tao_domain $DOMAIN &
```

We have provided an additional script, "*runall*" which starts all the Hosted Systems and SimpleDomainService.

To summarize, to run simpleexample the very first time, call *initkey*, *initdomain* and *inithost*. If no host is running, call *runhost*. Each time you run tests call *runall* but remember to kill these services afterwards.

Some further observations: The password supplied in the calls to `tao domain init` and `tao domain policy` protect access to the policy private key. The password supplied to the `tao newsoft`, `tao host init` and `tao host run` protect the soft host private key; this password is usually not the same as the key protecting the policy private key. `linux_host` which implements the Linux Host System and as the shell scripts are configured, uses the rules generated by `tao domain policy` to decide whether to run an application. Normally, the Host System does not use application security domain rules to determine what to run and, in fact, usually will run any application. This can be accomplished with the inclusion of the rule:

```
LinuxHost(x)
(forall P: forall T: forall H: TrustedHost(T) and LinuxHost(H) and Subprin(P, T, H) implies
TrustedLinuxHost(P))
(forall P: Program(P))
(forall P: forall Q: forall H: Program(Q) and TrustedLinuxHost(H) and Subprin(P, H, Q)
implies MemberProgram(P))
```

where x is the measurement of the linux_host or an “AllowAll” policy. As you become familiar with the Datalog rules, you can apply them flexibly but that is a distraction in SimpleExample.

What the output from SimpleExample teaches us about the Tao

The most concrete way to understand Cloudproxy is to follow the code example and the output. Here is a brief description of the output of the Go version of SimpleExample using a “soft” tao. In the execution setup, the domain information is in `/Domains/domain.simpleexample`; this includes the template, tao prepared configuration files and three directories: `SimpleClient`, `SimpleServer` and `SimpleDomainService` which are directories in which application information (mostly sealed keys) are stored for, respectively, `SimpleClient`, `SimpleServer` and `SimpleDomainService`. Binaries are stored in the directory `~/bin` as is customary in go.

In the repository, there are also three shell scripts to facilitate running the examples. The script compile compiles the applications and puts them into bin. After making the directory, `/Domains`, use *initdomainstorage* to initialize the storage areas. Copy the script clean into `/Domains/domain.simpleexample/SimpleDomain` and make it executable.

Thereafter, modify any code you wish to and run *compile* in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain` to compile the programs. ***Then*** *as root*, run *runall* to run SimpleExample. After it runs, you can run *clean*, in `/Domains/domain.simpleexample/SimpleDomain`, to erase the output files. *clean* runs a ps aux | fgrep simple at the end to tell you what lingering processes to kill (kill -9) so you can run subsequent tests.

Our example uses the Datalog authorization subsystem so system rules are expressed in the Datalog policy language. Example statements in Datalog can be seen in the template file.

When you look at the output, you'll notice, at the beginning:

```
Warning: Passwords on the command line are not secure. Use -pass option only for testing.
Warning: Passwords on the command line are not secure. Use -pass option only for testing.
Warning: Passwords on the command line are not secure. Use -pass option only for testing.
Warning: Passwords on the command line are not secure. Use -pass option only for testing.

Linux Tao Service (key([0801100118012245080112410431 0f3c0d7c5ff1...])) started and waiting
for requests

2016/02/20 11:27:13 simpledomainservice: Loaded domain
2016/02/20 11:27:13 simpledomainservice: accepting connections
```

This indicates that the linux_host, simpledomainservice, simpleclient and simpleserver have been initialized. The second section shows that the linux_host for the soft tao (with the indicated key) has started. The final section indicates that the domain service is started and waiting for request.

Next, you'll notice,

```
TaoParadigm: my name is
key([08011001180...]).Program([94d80d932fbc...]).key(f3169de17b1032dde230423f7d11dde89c14
3de147188fa67acf613d63da0420)
```

This is from *simpleserver*, and it is the Tao Principal Name of your *simpleserver* program, running on your Host System, after it has been extended with the hash of the loaded policy certificate. If you look at the source code for *simpleserver*, you'll notice that the policy key is not embedded in the code; if it had been, the policy key would be reflected in the program measurement. Instead, we read in the policy key cert and extend the *simpleserver* Tao Principal Name with the hash of the self-signed policy cert. The Tao Principal Name is hierarchical. The first segment, "`key([08011001180...])`", describes the host root[8]. The second segment, "`Program([94d80d932fbc...])`", describes the *simpleserver* program reflecting its measurement. The third segment, "`key(f3169de17b1032dde230423f7d11dde89c143de147188fa67acf613d63da0420)`", describes the policy key as noted above. Observe that the Tao Principal Name fully reflects all the program code as well as the policy it will execute (as represented by the policy key).

For the rest of this description, we will simplify terms like "`Program([94d80d932fbc...])`" as "`Program(`*`program-measurement`*`)`".

Next, notice the statement:

```
simpledomainservice, speaksfor: key(simpleserver_program_key) speaksfor key(host-
key).Program(simpleserver-measurement).key(policy-key)
```

This is the statement that TaoParadigm will use to request an attestation from the Linux Host System. The resulting Host System supplied attestation is

[8] If the root host had been a TPM, the name would include the TPM's AIK, and the contents of PCR 17 and 18 which contain the measurement of the booted Linux, extended with the initramfs which contains all the security critical files used by the Linux instance

key(*host-system-key*) from *notBefore* until *notAfter* says [*simpleserver-program-certificate*] speaksfor key(*linux-host*).Program(*simpleserver-measurement*).key(*policy-key*)

This statement is sent to the domain service which, after checking the measurements and domain policy signs a certificate (with pK_{policy}) that includes the statement

key(*policy-key*) from *notBefore* until *notAfter* says [*simpleserver-program-certificate*] speaksfor key(*linux-host*).Program(*simpleserver-measurement*).key(*policy-key*)

This is the *simpleserver* Program certificate. *Simpleserver*, as we described in the code annotations, stores this certificate, and sealed versions of the corresponding private *simpleserver* ProgramKey and SymmetricKeys. Decrypted and useable versions of these keys are populated in *serverProgramData* by *TaoParadigm*.

After initialization, *simpleserver* waits for client connections.

simpleclient meanwhile, goes through the same *TaoParadigm* initialization (which is not duplicated here) obtaining its Program certificate. *ServerClient* calls *OpenTaoChannel* with it's Program Cerificate and corresponding key. You'll notice, later in the output, that *simpleserver* opens a secure channel with a peer

key(*linux-host*).Program(*simple-client-measurement*).key(*policy-key*)

That peer is just your *simpleclient*. Normally, the Host System on which *simpleclient* runs will be different from the one *simpleserver* runs although in our case, they run on the same host.

Finally, you'll notice that *simpleserver* receives a request

```
2016/02/20 11:27:16     message type: 1
2016/02/20 11:27:16     request_type: SecretRequest
```

and returns the secret which is received by simpleclient as

simpleclient: secret iskey(linux-host).Program(simpleclient-measurement).key(policy-key)43

simpleclient encrypts and integrity protects the secret with its symmetric keys and the process concludes.

We have only discussed the major output elements here. Your output will contain much more including log messages from *simpledomainserver*.

The certificate for the simpleclient (which is in `/Domains/domain.simpleexample/SimpleClient/signerCert)` is:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1455996433984 (0x15300267640)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
        Validity
            Not Before: Feb 20 19:27:16 2016 GMT
            Not After : Feb 20 19:27:16 2017 GMT
        Subject: C=US, O=Google,
OU=key([0801100118012245080112410431 0f3c0d7c5ff1490ace20f167e7de1d5c6847c84d498c3b4a8087031
b49d9a38e7e59f4c5e4f23adc6ce2e394c7ac48923bcfcd7446bba0f86ef8bbdf89b6d5]).Program([d38d9410
0ae2bb57cccb97cb347ab060fb28c382bafe138f253fd19b491b1a15]).key(f3169de17b1032dde230423f7d11
dde89c143de147188fa67acf613d63da0420), ST=, CN=localhost
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
```

```
            pub:
                04:7a:d5:40:f6:80:fd:73:a5:80:b8:88:57:7c:60:
                6d:87:b6:78:4a:3f:fc:1c:cc:40:af:34:2d:98:31:
                02:21:02:71:65:66:7f:90:49:91:88:91:21:43:c7:
                f5:50:de:0a:7c:58:c8:6c:10:06:46:fc:3c:1a:a1:
                bb:c7:20:c6:83
            ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Key Usage: critical
                Key Agreement, Certificate Sign
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:0c:a0:99:55:79:0d:b7:26:20:07:38:03:da:ba:
         ff:28:0c:fd:94:f6:4e:5f:b1:ad:41:11:89:42:61:fd:5b:e7:
         02:20:7f:26:62:ee:2a:4c:90:e4:f4:7c:d6:c6:2c:b6:1d:db:
         d8:4a:bc:b9:60:26:aa:80:e8:bf:74:bd:ee:34:cb:fe
```

You'll notice that `/Domains/domain.simpleexample/SimpleClient/` also contains the files sealedsigningKey (*simpleclient's* sealed program private key), retrieved_secret (the "secret" encrypted with *simpleclient's* symmetric keys) and sealedsymmetricKey (*simpleclient's* sealed symmetric keys).

Running SimpleExample on a TPM1.2 machine

TODO

Upgrade and key management scenarios

Since sealed material is only provided to a Hosted System with exactly the same code identity that sealed the material running on the exact same Host System, while isolated by that Host System, you may be worried about lost data when a Hosted System breaks or becomes unavailable or limitations that may affect key management, software upgrade or distribution when the Hosted System runs on other Host Systems. In fact, it is rather easy to accommodate all these circumstances, and many others, efficiently, securely and in most cases automatically using Cloudproxy, although Cloudproxy applications must make provisions for this during development.

Below are a few sever example key management techniques that can be used when a Cloudproxy application is upgraded, a new Cloudproxy application (in the same security domain) is launched, or as applications migrate to other Host Systems. All these mechanisms preserve the confidentiality and integrity of all Cloudproxy applications and their data.

There is a discussion of many of the mechanisms, as they might affect client software used across different security domains, by users with no control over the application code while supporting consumer transparency (the most challenging case) in [4]. Here we restrict ourselves to cooperating server applications for simplicity.

To ease description, imagine all application data is stored locally or remotely and probably redundantly in encrypted, integrity protected files. Each file is encrypted and integrity protected

with individual file keys and each file key is itself encrypted and integrity protected with a group sealing keys. Different groups of file keys are protected by different sealing keys to reduce the risk of universal compromise. Every key has exposed meta data consisting of a globally unique name for the entity it protects, the key type and an "epoch." Epochs increases monotonically as the keys are rotated[9]. As keys for a new epoch become available, the objects they protect are re-encrypted, over a reasonable period of time (the Rotation Period). During this time, keys for the prior epoch are available and can be used to decrypt objects; however, as soon as new epoch keys are available, all new data is encrypted with the new epoch keys. At the end of the Rotation Period, once applications have confirmed that all data is protected with the keys from the most recent epoch, old epoch keys are deprecated.

The first option to deal with "brittle keys" protecting application data is standard: use a distributed key server like Keyczar (or many others). In this case, Cloudproxy applications do not locally store data protection keys but contact a key server (over a Tao Channel). The key server (which does key rotation, etc., as many do) authenticates the Hosted System that needs keys and verifies that it is authorized to receive those keys; if so they are transmitted over the Tao Channel. Hosted Systems can be upgraded and all authorization policy can be maintained by the key service. Hosted Systems will need to respond to "reinitialize" requests periodically as keys rotate.

An alternative, less centralized, key rotation mechanism allows individual Hosted Systems maintaining their own keys to protect files as well as perform key rotation themselves. When software is upgraded or new programs are introduced, the new programs or upgraded programs come with a certificate signed by the policy key that instruct one Hosted System to disclose these keys to the new version (or new) Hosted System. Since this can result in lost data if a Host System becomes unavailable, Hosted Systems would likely distribute these keys to different instances on different machines to ensure continuity.

Finally, when new data protection keys are established for an application task, Hosted Systems can contact a domain service to receive intermediate keys for registered files or file classes. These keys can be sealed using the Host System provided Seal and used without contacting the service each time the Hosted System starts. This mechanism places additional administrative burden on each Hosted System to contact the "key sharing service" as intermediate keys rotate but this is not uncommon.

It is important to note that while the foregoing descriptions treat keys as "all or nothing" entities, all these scheme have corresponding "split key" implementations to achieve higher security. In addition, any security domain may elect to have an authorized Cloudproxy Hosted System archive data. Such an archive application, upon which security domain policy confers access to data, can, in the background, archive data to (centralized or distributed) repositories.

As a reminder: there are other possible mechanisms to do key management.

[9] And you certainly should rotate keys as part of effective cryptographic hygiene!

Finally, note that application upgrade (given a data key management solution) is automatic even when the policy keys change: New versions of Hosted Systems simply re-initialize (get new program keys and certificates) using the (centralized or distributed) security domain service and no special provision, aside from current policy at the security domain service, need be provided[10].

Suggested Exercises

That's all there is to using Cloudproxy. Here are some suggested exercises to complete the training:

1. Write a more complicated set of domain applications; for example, see "`$CLOUDPROXYDIR/go/apps/fileproxy`."
2. Boot a Linux Host System on tpm supported hardware using the TPM to root the Linux Tao (see … for instructions).
3. Boot a KVM Host System on tpm supported hardware and then run a stacked VM host in a Linux partition (see … for instructions). SimpleExample should run fine in the VM(s) with slight changes to the initialization scripts.
4. Explore the Data log engine (examples?)
5. What happens if you make a modification to simpleclient.go and immediately run it on `linux_host` without reinitializing the domain?

References

[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing, http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.
[2] CloudProxy Source code, http://github.com/jlmucb/cloudproxy.
[3] TCG, TPM specs,
http://www.trustedcomputinggroup.org/resources/tpm_library_specification
[4] **Beekman, Manferdelli, Wagner,** AsiaCCS, 2016.

[10] Many events may cause such a policy change including a determination that previously trusted hardware elements have been compromised.

The Guard

The Guard interface:

- `Subprincipal() auth.SubPrin`: returns a unique subprincipal for this policy.
- `Save(key *Signer) error`: writes all persistent policy data to disk, signed by key
- `Authorize(name auth.Prin, op string, args []string) error`
- `Retract(name auth.Prin, op string, args []string) error`
- `IsAuthorized(name auth.Prin, op string, args []string) bool`
- `AddRule(rule string) error`
- `RetractRule(rule string) error`
- `Clear() error`: removes all rules.
- `Query(query string) (bool, error)`
- `RuleCount() int`
- `GetRule(i int) string`.
- `String() string`: returns a string suitable for showing auth info.

Tao Go API

```
type Tao interface {
        // GetTaoName returns the Tao principal name assigned to the caller.
        GetTaoName() (name auth.Prin, err error)

        // ExtendTaoName irreversibly extends the Tao principal name of the caller.
        ExtendTaoName(subprin auth.SubPrin) error

        // GetRandomBytes returns a slice of n random bytes.
        GetRandomBytes(n int) (bytes []byte, err error)

        // Rand produces an io.Reader for random bytes from this Tao.
        Rand() io.Reader

        // GetSharedSecret returns a slice of n secret bytes.
        GetSharedSecret(n int, policy string) (bytes []byte, err error)

        // Attest requests the Tao host sign a statement on behalf of the caller. The
        // optional issuer, time and expiration will be given default values if nil.
        // TODO(kwalsh) Maybe create a struct for these optional params? Or use
        // auth.Says instead (in which time and expiration are optional) with a
        // bogus Speaker field like key("") or nil("") or self, etc.
        Attest(issuer *auth.Prin, time, expiration *int64, message auth.Form)
            (*Attestation, error)

        // Seal encrypts data so only certain hosted programs can unseal it.
        Seal(data []byte, policy string) (sealed []byte, err error)

        // Unseal decrypts data that has been sealed by the Seal() operation, but only
        // if the policy specified during the Seal() operation is satisfied.
        Unseal(sealed []byte) (data []byte, policy string, err error)
}

// Parent returns the interface to the underlying host Tao. It depends on a
// specific environment variable being set. On success it memoizes the result
// before returning it because there should only ever be a single channel to the
// host. On failure, it logs a message using glog and returns nil.
// Note: errors are not returned so that, once it is confirmed that Parent
// returns a non-nil value, callers can use the function result in an
// expression, e.g.:
//   name, err := tao.Parent().GetTaoName()
func Parent() Tao {
        ParentFromConfig(Config{})
        return cachedHost
}
```

SimpleExample in C++

Simple Client in C++

Simple Server in C++